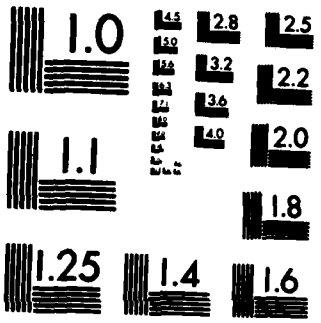
AD-A127 338 STEP-WISE DEBUGGING(U) MARYLAND UNIV COLLEGE PARK DEPT 1/1
OF COMPUTER SCIENCE D HAMLET DEC 82 TR-82/16
AFOSR-TR-83-0309 F49620-80-C-0001

UNCLASSIFIED F/G 9/2 NL

END
DATE FILMED
5 83
DTIC



(12)

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF MELBOURNE

STEP-WISE DEBUGGING

Dick Hamlet

Technical Report 82/16
December. 1982

Department of Computer Science
University of Melbourne+
Parkville, Victoria 3052
Australia

Abstract

Programs written using step-wise refinement are meant to be understood at levels of an hierarchy, but only a complete program can be executed. This paper considers the debugging of such programs in the manner they are written: top-down, each level not depending on execution of the levels below. Step-wise debugging also applies to the testing of multiple-process systems, and to the integration testing of any software. It allows the debugging of incomplete software, and when all modules are available, it structures debugging to follow the conceptual organization of the software.

+On leave from Department of Computer Science, University of Maryland, College Park 20742 U S A. This work was partially supported by the United States Air Force Office of Scientific Research under grant F49620-80-C-[redacted]

0001 -a-

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-0309	2. GOVT ACCESSION NO. AD-A127 838	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) STEP-WISE DEBUGGING		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL 6. PERFORMING ORG. REPORT NUMBER TR #82/16
7. AUTHOR(s) Dick Hamlet		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE DEC 82 13. NUMBER OF PAGES 10
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Programs written using step-wise refinement are meant to be understood at levels of an hierarchy, but only a complete program can be executed. This paper considers the debugging of such programs in the manner they are written: top-down, each level not depending on execution of the levels below. Step-wise debugging also applies to the testing of multiple-process systems, and to the integration testing of any software. It allows the debugging of incomplete software, and when all modules are available, it structures debugging to follow the conceptual organization of the software.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. Decomposition of Programs

"Divide and conquer" is acknowledged to be the best strategy for combating the immense complexity of computer programs. In its least structured form, this philosophy involves dividing a program into "modules," each so small that it can be understood easily. Unfortunately, a bad division can be counterproductive in that the data interfaces between modules are so broad, and the control connections so chaotic, that understanding of the whole does not come from understanding of the parts. Two methods are in common use to structure decomposition: hierarchical step-wise refinement, and cooperation of autonomous processes. Each of these is an imprecise programming method whose spirit is not necessarily captured by the letter of its syntactic laws.

Step-wise refinement [1] is usually applied to single, sequential programs; then the modules are conventional procedures. The call graph of dependencies among modules has well-defined levels of limited complexity. (The procedures are short and do not call too many others; the call graph has limited breadth. Recursion--cycles in the graph--is usually thought to be beneficial.) The idea is that to understand a node of the graph requires only understanding of its immediate descendent nodes, and understanding the root node is mastery of the whole. The decomposition is logarithmic in that to handle an N-module system requires dealing with only about log N modules at a time.

The other major method for controlling decomposition uses units that are autonomous programs (processes) instead of subroutines, communicating by passing messages to each other [2]. The analogy of the call graph is a graph whose nodes are processes and whose arcs are message-interchange connections. Compared to the case of subroutine modules, the nodes are more complex, but the connections simpler. In the extreme case represented by UNIX pipes [3] the graph is a chain, and the nodes may be sophisticated programs. It is more usual to allow slightly more complex communication, and to restrict the complexity of each process. The idea is that each process can be understood alone, and the behavior of the whole is obvious given that of the parts. Because communication is asynchronous and each process complex, the interconnections must be very simple to realize this goal. Ideally, understanding an N-module system requires only dealing with a fixed number K ($K << N$) of modules at a time.

During development of a system using one of these methods, each of the units can be assigned to a separate designer or programmer, who need be given only information about immediately connected units. Since this information is necessarily less precise than actual design or code for the other units, it is difficult to debug units in isolation. The most common schemes are to wait for all units to be completed, and test them in a bottom-up fashion so that all communications are supplied as they will actually be in the complete system, or to test top-down using stubs that do not behave much like the missing modules, but merely (say) print messages announcing that they have been called. This paper considers how to do a better job of top-down debugging. The essence of the idea is iteration through the decomposition links: the information that should have been supplied by missing modules is given at random in stage 0, creating outputs that are used as inputs in stage 1, and so on until the process converges (if it does) to a useful test.

Step-wise debugging requires the support of a bookkeeping tool, because for a large system the volume of data and its proper labeling is difficult for a person to handle. In some cases it is useful for a human user to monitor

details of the debugging process; more often it is desirable that all its details be hidden, for which machine assistance is essential.

2. Step-wise Debugging of Hierarchically Organized Subroutines

There is good reason to deal with a system one module at a time even when all modules are available. In a complex, just-completed system, bugs are likely to lurk everywhere, and letting the modules invoke each other as if the system were working is a waste of time: the results are too chaotic to understand. Step-wise debugging can use an incomplete set of modules; it is easiest to describe the method for the case in which all modules are available. (The more complex case of cooperating processes is treated in Section 3.)

Suppose then that a complete collection of modules is available, to be debugged one at a time. The first may be handled as in a conventional top-down integration test [4], with the results from other modules supplied by random selection. When the test is complete it yields a collection of inputs sent to other modules, and one input-output pair for the test module. As each additional module is tested, inputs given to other modules accumulate, and input-output pairs. By working from the list of inputs so generated, and using already-available pairs when possible, the reliance on randomly generated results can be decreased. A series of debugging stages can be identified for each input to the entire system. At stage 0 each module is tested in isolation, using randomly generated values for all intermodule communications, except where a result has already been obtained at stage 0. At stage 1, each input for another module whose result was needed in stage 0 is investigated. The necessary routine is tested with this input, using input-output pairs from stage 0 (or, in preference, stage 1) where possible, otherwise random generation of results. As each value is obtained at stage 1, any previous calculations that could use it are repeated, perhaps adding to the list of needed results. Continuing in this way, at stage N there is a list of inputs for various modules whose results were needed but unavailable at stage N-1. These are used as tests, utilizing the most recent results where possible, and the outcomes substituted in the most recent earlier computations, resulting perhaps in further corrective substitutions, and creating the needed list for stage N+1.

The step-wise debugging process can be carried out without human intervention except to supply a system input and to select the "top" module for test. Since the results depend on the order in which modules are considered, the call graph can be used to completely describe the system structure. For a collection of modules without recursion, the process necessarily terminates, because the call graph has no cycles, and hence some modules do not depend on others. Their results in turn fix the results from modules higher in the graph, and a stage exists in which only information from actual executions is being used, so that no new input list is generated. At the termination stage, the execution is indistinguishable from a conventional bottom-up integration test of the software. But debugging is directed from the top, which shows itself in the tests applied at lower levels. (Very similar remarks could be made about top-down development: when it is complete, the design can be viewed bottom-up, but it could not have been constructed that way.)

Step-wise debugging can be used on a partial system if information is available about the missing modules. The interface parameters must be strongly typed (the more restrictions the better) and marked as inputs or outputs. (This allows random selection of results from an appropriate range.) An informal specification must be available. (It is used to check that

supplied results are reasonable.) The debugging process then terminates at stage 1 unless there is recursion.

When there is recursion the calls can be allowed to invoke the code instead of being stopped at each stage. (Allowing recursions to occur can be compared to allowing loops within the code to proceed without interference. For cooperating processes, loops themselves must be controlled, as described in Section 3.) The recursion may terminate, and the debugging will then also end. If the human user is impatient, the specification may be used to supply a value at stage N; if the corresponding input is not added to the list for stage N+1, debugging will terminate. Both techniques can also be used when all modules are available: allowing code to be used instead of enforcing the stage boundaries corresponds to introducing a measure of bottom-up testing; supplying values that are close to correct can speed up convergence. (If values supplied at stage N disagree with the actual computations, it will show up at stage N+1.)

3. Step-wise Debugging of Cooperating Sequential Processes

A module-connection graph with process nodes, and arcs indicating message interchange, does not capture the cooperation of processes as it usually occurs in so-called real-time systems. In such systems each process is cyclic (perhaps implemented using a never-ending loop) with its body devoted to interacting with other processes and performing calculations based on those interactions. The process "output" is the messages it sends to other processes, and its "input" is received from them. The external world can be thought of as another process interacting with the computer system. (This view is due to Fitzwater [5] and has been used as the basis of a specification language [6].) Processes of this kind are not functional in the sense of transforming inputs to outputs, but they can be debugged in the step-wise manner described in Section 2 if the unending loop is interrupted at each debugging stage. (The loop introduces "recursion" into the interconnection graph which might be imagined as each process reinvoking itself as soon as its body is completed.)

Another important feature of cooperation among processes is the synchronization of communication, which can be incorporated in message passing by allowing a process to test for a waiting incoming message. Such a test can be handled as an interchange with the sending process whose values come from a {wait, nowait} set. Communication occurs as follows: (1) processes communicate by name; (2) process output is immediate; (3) process input waits for the corresponding output; (4) a process may test for input waiting. To combine processes so that their messages could be exchanged during actual execution (at later stages of the debugging process), it is sufficient to have a time stamp associated with each message and input-waiting test.

When all processes are available, step-wise debugging proceeds as follows. At stage 0, each process body is executed in isolation, with incoming messages randomly generated (but see below), and decisions about waiting input treated as two-valued random choices. Each message and choice is stamped with the execution time of the process at which it occurs. To determine if any previously obtained stage-0 outputs can serve as inputs in place of randomly generated ones (and similarly when later stages use earlier results) requires analysis of the message and test time stamps. If process S sends a message to process R stamped t, it can be used for input at time u only if all of R's input-waiting tests on S before t and before u have failed, and all those after t and before u have succeeded. Should such an output be used, as

input it is time stamped the greater of t and u , reflecting the fact that R had to wait for it. Should t be greater than u , execution time for R is also updated to t . At stage 1, the lists of input needed by each process are considered. For each output message needed from process P, P is executed using stage-0 and stage 1 inputs where possible, until one is produced. (This may require more than one repetition of P's body, and indeed the necessary message may never be forthcoming, as discussed below.) When a message is obtained, the calculation in which it is involved as input is repeated, including the necessary revisions in time stamps discussed above. This may alter the list of needed messages. Continuing in this way, at stage N there is a list of messages needed but not supplied by stage N-1, which are sought using the most recent messages where possible, then substituted into the most recent earlier computations. This may force yet earlier substitutions, and creates the message-needed list for stage N+1.

To illustrate debugging of cooperating processes, something like a Pascal program will be used for each process, with the message-interchange constructions:

```
send(link, message)
```

meaning that output of "message" is to occur to process "link"; and,

```
receive(link, variable)
```

meaning that execution must wait for a message from process "link" which is placed in "variable"; and,

```
ready(link)
```

a Boolean function that is true iff process "link" has sent a message that the executing process may now receive . The identifier *EXTERNAL* is a "link" naming the environment that is not part of the cooperating system, to which these constructions may be applied for input-output.

Consider the following collection of processes, which is typical of simple systems in which different routines handle different kinds of input in parallel, but with some interaction. The system is intended to keep a check on the number of input "0" and "1" characters, and to respond upon "-" with whether or not the prior counts were the same.

```
process Classify;
var letterin: char;
begin
   while true do
      begin
         receive(EXTERNAL, letterin);
         case letterin of
            '0': send(Zero, letterin);
            '1': send(One, letterin);
            '-':
               begin
                  send(One, letterin);
                  send(Zero, letterin)
               end
         end
      end
end.
```

```
process Zero;
var Z: char;
    Excess: integer;
begin
   while true do
      begin
         receive(Classify, Z);
         if Z = '-' then
            begin
               send(One, '+');
               Excess := 0;
               while Z <> '+' do
                  begin
                     receive(One, Z);
                     Excess := Excess + 1
                  end;
               send(Advise, Excess)
            end
         elseif ready(One) then
            receive(One, Z)
         else
            send(One,Z)
      end
end.

process One;
var Z: char;
    Excess: integer;
begin
   while true do
      begin
         receive(Classify, Z);
         if Z = '-' then
            begin
               send(Zero, '+');
               Excess := 0;
               while Z <> '+' do
                  begin
                     receive(Zero, Z);
                     Excess := Excess + 1
                  end;
               send(Advise, Excess)
            end
         elseif ready(Zero) then
            receive(Zero, Z)
         else
            send(Zero,Z)
      end
end.

process Advise;
var A, B: integer;
begin
   while true do
      begin
         receive(One, B);
         receive(Zero, A);
         send(EXTERNAL, A = B)
      end
end.
```

Applying the algorithm above to the input

0010-1

at stage 0, we have:

time	process	message: from	to	value
	Classify			
3			Zero	'0'
6			Zero	'0'
9			One	'1'
12			One	'-'
13			Zero	'-'
16			One	'1'
	Zero			
2		Classify		'0'?
4		One		ready=true?
5		One		'+'?
7		Classify		'0'?
9		One		ready=false?
10			One	'0'
12		Classify		'-'?
15			One	'+'
18		One		'+'?
21			Advise	1
	One			
2		Classify		'1'?
4		Zero		ready=false?
5			Zero	'1'
7		Classify		'-'?
9			Zero	'+'
12		Zero		'+'?
15			Advise	1
	Advise			
2		One		3?
3		Zero		8?
4			EXTERNAL	false?

Time is measured in statement counts. The randomly generated values are followed by a question mark (?), and are selected from the range of characters that could be determined by static analysis, or from the integer interval [0,9]. Several cycles of the processes Zero and One have been performed; how far to go is determined by the needs of stage 1 (here for process Advise).

The stage-0 approximation to this system's behavior is not very good, and it should be considered an accident that the desired output results--it would also appear for the input

01-

by the same analysis. However, we know that the process is not finished because some messages were generated at random.

At stage 1 there are no messages needed by Classify. Process Zero needs a message from Classify, which has appeared in stage 0 at time 3, so the time within Zero is updated, but the randomly generated message value of '0' stands. The stage-0 analysis of One shows that the ready query would be false at time 4, so the last two stage-0 messages in Zero are replaced. A similar analysis of the other routines gives the partial stage-1 result:

time	process message: from	to	value
	Zero		
3	Classify		'0'
5	One		ready=false[0?]
6		One	'0'
	One		
9	Classify		'1'
11	Zero		ready=true[0?]
12	Zero		'+'[0?]
	Advise		
16	One		1[0?]
22	Zero		1[0?]
23		EXTERNAL	true

The "0?" following a message value indicates that a result for stage 0 has been used that is itself based on a randomly generated message. Hence these values will have to recalculated later.

Continuing in this way, the final result is:

time	process message: from	to	value
	Classify		
3		Zero	'0'
6		Zero	'0'
9		One	'1'
12		One	'-'
13		Zero	'-'
16		One	'1'
	Zero		
3	Classify		'0'
5	One		ready=false
6		One	'0'
8	Classify		'0'
	One		ready=false
		One	'0'
14	Classify		'-'
16		One	'+'
19	One		'+'
22		Advise	1

```
    One

 9                  Classify            '1'
11                  Zero                ready=true
12                  Zero                '0'
14                  Classify            '-'
16                                Zero  '+'
19                  Zero                '0'
22                  Zero                '+'
24                                Advise 2
26                  Classify            '1'

    Advise

25                  One                 2
26                  Zero                1
27                                EXTERNAL false
```

(It might be useful to reproduce this table in response to the input, but it is easier to understand if the process actions are intermingled on a single time scale.)

For this expository example, the result is the same as if the processes had been allowed to "just run" under a reasonably fair scheduling algorithm, and the resulting table is similar to that suggested in [7]. For more complicated cases, however, the ordered consideration of one process at a time distorts the scheduling decisions in favor of that process. Two useful outcomes are that (1) since processes "get what they want," when they contain simple errors, the repair may be easier; and (2) by altering the order in which processes are considered, unusual scheduling patterns can be obtained, without a user having to describe those patterns in detail. A buggy routine may even be characterized by wrong output when it is placed at the top level in stepwise debugging, but correct results when it is relegated to a subordinate position.

External inputs may be generated at random, in units of the size messages needed. This further emphasizes processes placed at the top level, since an input process does not use its time sequence to order the scheduling (as occurred in the sample above). Then step-wise debugging may terminate even though execution of the processes does not. That is, a stage may be reached at which no new messages are needed. This means that a particular sequence of interactions has been discovered that is time-consistent, and so represents a possible actual execution of the system. The sequence obtained will depend on the order in which modules are tried, which can reflect a hierarchical view of the system on top of its process structure. A terminating debugging session yields a system test in which each process body is executed an integral number of times, and there are no missing inputs. Continuing to execute any process may begin a new (possibly different) cycle.

When a process P should produce a message at stage N, it can fail, either because the message cannot be produced, or because P has not itself been given the proper messages to yield the needed result. In the former case the debugging process cannot continue until P is repaired; in the latter case it may be sufficient to abandon P at stage N, and add the needed message to stage N+1, where better data may be available.

For a partial system, two-stage debugging requires that relative timing information be supplied, as well as messages from the missing processes. This puts a considerable burden on the specifications for those processes, but in a language designed for partial specification the process can succeed, as

described in [7].

4. Computer Support

The example of step-wise debugging given in Section 3 shows the need for automatic bookkeeping to handle large collections of input-output data, and to keep track of the method's stages. When portions of the code are allowed to "just run," and when results are artificially inserted for missing routines, conventional debugging and test-harness tools are appropriate. For example, a concurrent-control language [8] would help monitor a process whose output was needed.

A support implementation is straightforward. The techniques of self-contained interpreter, preprocessor, or run-time library with compiler modification are all applicable. For a conventional programming language like Ada (and for most program design languages) the run-time library approach can be easily added to a conventional debugging system with procedure-call tracing. Since the trace routine gets control at each call, it can be modified to generate values and record the needed-list. Instead of allowing the program itself to execute, a phony main program is added that invokes the instrumented procedure code, in accordance with the stages of the method. Essentially the same techniques have been used for an automatic testing system [9].

The case of cooperating processes will be described in more detail. Consider a systems implementation language with operating-system support for process control and message interchange. This is the approach used in C under UNIX, and in many "quick and dirty" languages like SIMPL-XI [10]. These can serve as object languages for the compilation of higher-level specification and requirements source languages, so they represent the general case: minor changes to the high-level compiler create a support system at that level. For concreteness, assume that a system uses C with UNIX support. (That message-passing is very inefficient in current versions of UNIX will not disturb understanding.) In execution, systems appear to be a collection of stand-alone programs, which can communicate through operating-system calls.

The debugging system gains control within each process by intercepting operating-system service calls. For example, instead of processes calling SIGNAL to use traps for communication, they call a new library routine PRESIGNAL, which calls SIGNAL after appropriate bookkeeping has been done. Then traps go to PRESIGNAL, which can keep track of the communication as required. For debugging execution, control is not given to any of the processes. Instead, a library process PREMAIN initially assumes control, and obtains the information about process ordering from its user. Instrumented versions of the processes to be debugged are started by PREMAIN using FORK. The list of processes and input is obtained from the user at stage 0; thereafter the lists from previous stages are used. Any message activity is caught by PRESIGNAL. At stage N, PRESIGNAL supplies the previously recorded or randomly generated values from other processes without invoking them, accumulates the lists for stage N+1, and when an adequate cycle has been completed, returns control to PREMAIN.

5. Summary

A method of step-wise debugging has been described which structures the usual chaotic process of testing a large piece of software. The resulting test runs

are ones that could have been performed without the method, but probably would not have been. The software structure directs the selection of interface test values in a top-down way. The method suggests a straightforward computer tool to support the extensive bookkeeping required.

References

1. N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

2. C. A. R. Hoare, Communicating sequential processes, CACM 21 (Aug., 1978), 666-677.

3. D. M. Ritchie and K. Thompson, The UNIX time sharing system, CACM 17 (July, 1974), 365-375.

4. G. J. Myers, The Art of Software Testing, Wiley, 1979.

5. D. R. Fitzwater, A decomposition of the complexity of system development processes, COMPSAC 78, Chicago, 424-429.

6. P. Zave, An operational approach to requirements specification for embedded systems, IEEE Transactions on Software Engineering SE-8 (May, 1982), 250-269.

7. P. Zave, Testing incomplete specifications of distributed systems, Proc. ACM Symposium on Principles of Distributed Computing, Ottawa, 1982, 42-48.

8. P. C. Bates and J. C. Wileden, EDL: a basis for distributed system debugging tools, Proc. 15th Hawaii International Conference on System Sciences, Honolulu, 1982, 86-93.

9. R. G. Hamlet, Testing programs with the aid of a compiler, IEEE Transactions on Software Engineering SE-3 (July, 1977), 279-290.

10. R. G. Hamlet and M. V. Zelkowitz, SIMPL systems programming on a minicomputer, Digest of Papers, 9th Annual IEEE Computer Society International Conference, Washington, 1974, 203-206.